Computer Science • 24(4) 2023 https://doi.org/10.7494/csci.2023.24.4.5539

Anna Gajos-Balińska
Grzegorz M. Wójcik
Przemysław Stpiczyński

HYBRID IMPLEMENTATION OF THE FASTICA ALGORITHM FOR HIGH-DENSITY EEG USING THE CAPABILITIES OF THE INTEL ARCHITECTURE AND CUDA PROGRAMMING

Abstract *High-density electroencephalographic (EEG) systems are utilized in the study of the human brain and its underlying behaviours. However, working with the EEG data requires a well-cleaned signal, which is often obtained using independent component analysis (ICA) methods. The longer the calculation time for these types of algorithms is, the more data is obtained. This paper presents a hybrid implementation of the fastICA algorithm that uses parallel programming techniques (libraries and extensions of the Intel processors and CUDA programming), which results in a significant acceleration of execution time on selected architectures.*

Keywords ICA, EEG, BLAS, MKL, OpenMP, Intel Cilk Plus, CUDA

Citation Computer Science 24(4) 2023: 455–472

1. Introduction

EEG systems are commonly used in research on brain function as well as to investigate the neural basis of behaviour or cognition. Unlike the other brain imaging techniques such as functional magnetic resonance imaging (fMRI) or positron emission tomography (PET), which have a relatively low spatial resolution, high-density EEG systems can provide an accurate view of brain activity, with an electrode resolution of up to several hundred sensors. These systems have a wide range of applications in such fields as neuroscience, psychology, and medicine [22, 24–27]. However, thorough cleaning of the signal of artifacts (parts of the signal that do not originate from the brain) is crucial for its further analysis, however, this process can be time-consuming [1, 4, 13]. Additionally, manufacturers of EEG systems do not always meet the expectations of researchers and the methods available in their tools are not satisfactory enough. The most frequently used methods of signal purification are ICA-type algorithms [2, 3, 11, 23] and they are most often available in other external programs, not adapted to larger batches of data. ICA is a powerful tool for analyzing and interpreting complex data sets and has a wide range of applications in such fields as machine learning or data mining. However, due to the iterative nature of ICA-type algorithms, their computation time can be daunting.

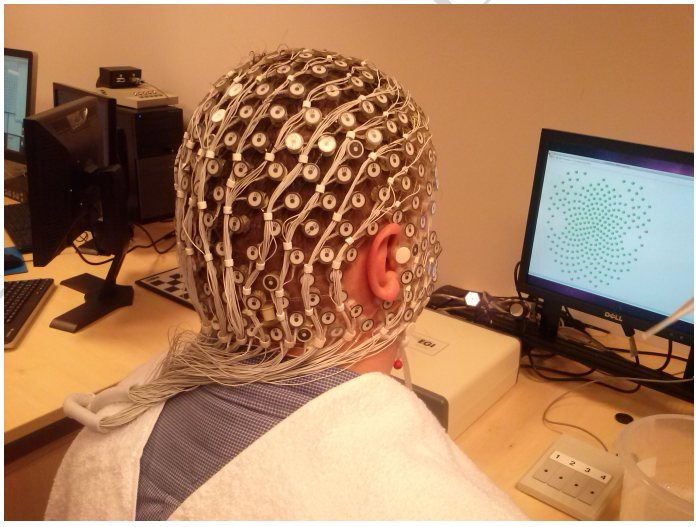


Figure 1. 256-electrode cap in EEG Laboratory

For example, for a256-electrode EGI system with 1000 Hz sampling [17], it can take several hours to process a 10-minute study. This paper describes an implementation that accelerates these calculations by exploiting the capabilities of multi-core architectures. Initially, the implementation was based on parallel calculations on CPU cores using the Intel processors, and over time it was enriched with matrix calculations

using the CUBLAS library, which is available through programming in CUDA for the NVIDIA graphics cards [7–10]. The paper presents improvement that transfers more calculations to the graphics card, thus reducing the calculation time even further. The calculation time obtained in this way was compared with the implementation time without the CUDA support.

2. Materials and methods

2.1. Independent component analysis

Independent component analysis (ICA) is a statistical technique used to separate independent sources that affect data. In the context of EEG the set of independent components can be interpreted as the sources of brain activity that generated the recorded signal. ICA belongs to a family of methods called blind source separation (BSS) [2].

The BSS problem can be represented by the equation:

$$\mathbf{S} = \mathbf{W}\mathbf{X}, \tag{1}$$

where $\mathbf{S} \in \mathbb{R}^{CxM}$ is the matrix of C components for M samples, $\mathbf{W} \in \mathbb{R}^{CxN}$ is the transition matrix with the weight vectors between each signal and electrode and $\mathbf{X} \in \mathbb{R}^{NxM}$ is the data from N electrodes. W is the unknown separation matrix that will satisfy this equation.

The limitation of the BSS methods consists in impossible determination of the original amplitude of the source signals and no more than $\mathbb{N}$ sources can be found for $\mathbb{N}$ recorded signals [12]. Additionally, independent component analysis (ICA) relies on the assumption that the sources are statistically independent, indicating that the sources are not correlated with each other and as a rule not distributed in the value domain. If a distribution of the original signals is close to normal, the result can be ambiguous [12]. The goal with a matrix $\mathbb{S}$ is to find the components of $\mathbb{X}$ that are as independent of each other as possible. To obtain this, the data is pre-processed, specifically centered (the mean of each signal is zero) and whitened (variance of each signal is equal to 1) to remove correlation. Then, different measures of normality (negentropy or kurtosis) are used to modify a $\mathbb{W}$ matrix using the Newton approximation method and the chosen non-quadratic function.

There are many variations of the ICA algorithm. However, the fastICA algorithm was chosen for the implementation because it is the most commonly used and it was easy to use parallel methods on it.

2.2. Data representation and implementation

The fastICA algorithm implementation was written in C, based on the version available in Matlab and the open-source it++ library. The *tanh* (hyperbolic tangent)

function from the basic mathematical library was used to modify the weights. A symmetrical approach of searching for components was adopted, indicating that the algorithm calculates all components simultaneously.

It is worth noting that fastICA in the it++ and Matlab implementations use matrix reduction to analyze multidimensional data, which reduces computational accuracy. However, it reduces computational complexity. According to the Matlab documentation, the fastICA algorithm for large data sets is not stable. The data set obtained in the EEG study is multidimensional, but the analysis of the results should be based on those most accurate. Therefore, the solution does not reduce the matrix dimension and calculations are performed for the entire data set. The most important aspect of the solution was the development of a methodology for time optimization of individual steps of the algorithm based on the use of the capabilities of multi-core architectures. This implementation applies the OpenMP interface.

In Listing 1 there is a scheme of fastICA algorithm with an indication of the most important steps. The *mul* functions use matrix multiplication on the CPU or GPU depending on the version of the implementation.

Listing 1: Scheme of fastICA algorithm

```
1  // W - separating matrix
2  // n - number of eletrodes, m - number of samples
3  // resultW - the resulting separation matrix
4  // whiteningMatrix - calculated earlier
5    randWeight(W, n, n); // random weights
6    orth(W, n, n); // orthogonalization
7    while (!found) { // next iterations
8      found=0;
9      if (it > maxNumIterations - 1) {
10       //iteration count exceeded
11     }
12     it++;
13     // Weight matrix normalization and convergence estimation
14     // Checking if the weights have changed
15     if(!found) {
16       // Modification of weights and set found parameter
17     }
18   }
19   // Rewriting the final form of the separating matrix
20   mul(W, n, true, whiteningMatrix, n, false, n, resultW);
```

The *parallel* directives from OpenMP were used in parts of the algorithm where the entire signal was applied (signal whitening and calculation of successive approximations). With the Intel Cilk Plus extensions for C and C++, one can use array notation and built-in reduction functions (such as finding the maximum or minimum value in an array), which makes the code more readable and forces effective vectorization. For parts of the algorithm, such as matrix multiplication, vector and eigenvalue

calculations, it was much more advantageous to use ready-made solutions from the BLAS and MKL libraries (*cblas_dgemm*, *cblas_dcopy*, *LAPACKE_dsyev*, *LAPACKE_dgesvd*) than their implementations [18].

Originally, in the iterative part of the algorithm, the matrix multiplication of the entire signal was implemented by multiplication functions from the CUBLAS library, which is the equivalent of the BLAS library for the NVIDIA graphics cards. When performing calculations on the graphics card, it is important to be aware of the additional time overhead associated with host-device data transfer [28]. In this situation, each iteration means additional operations related to uploading and downloading data. Using the potential of the GPU inside the iterative loop, each data transfer seemed risky. However, by transferring the calculations related to the use of the non-square function and modifying the weights to a separate library compiled by nvcc, the size of the data sent at a time was decreased. This required writing a function dedicated to the graphics card (kernel) taking into account the problems of memory shared by blocks and threads of the CUDA architecture. In Listing 2, you can find a code fragment along with its invocation. On the other hand, in Listing 3, there is a section of C code that executes in each iteration.

Listing 2: Code for kernel function

```
__global__ void kernelSum(double * __restrict__ A,
    double * __restrict__ blockResults, int m, double a1) {

  extern __shared__ double sums[];
  double sum = 0.0;
  double * p = &A[blockIdx.x * m];

  for(int i = threadIdx.x; i < m; i += blockDim.x) {
    // Using of tangent functions
    A[blockIdx.x * m + i] = tanh(a1*p[i]);
    sum += (1 - pow(p[i], 2));
  }

  sums[threadIdx.x] = x;
  __syncthreads();
  for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
    if(threadIdx.x < offset) {
      sums[threadIdx.x] += sums[threadIdx.x + offset];
    }
    __syncthreads();
  }

  if(threadIdx.x == 0) {
    blockResults[blockIdx.x] = sums[0];
  }
}
```

```
27 void tanhGPU_wrapper(double * A, double a1, int n, int m,
28     double * X, double * W, double * G, double * nvec) {
29
30   int nm = n*m;
31   int bs, mings;
32
33   cudaOccupancyMaxPotentialBlockSize(
34         &mings, &bs, tanhGPU, 0, nm);
35   size_t shmsize = sizeof(double) * bs;
36
37   mulGPU(X, m, true, W, n, false, n, A);
38
39   kernelSum<double><<<n, bs, shmsize>>>(A, nvec, m, a1);
40
41   mulGPU(X, n, false, A, n, false, m, G);
42
43   cudaDeviceSynchronize();
44 }
```

Listing 3: Part of the code with the modification of weights using the kernel function

```
1  // W - buffer with weights
2  // c_W - buffer with weights for CUDA
3  cublasSetVector(nn, sizeof(*W), W, 1, c_W, 1);
4
5  // n - size; W is matrix nxn
6  copyMatrix(Wprev, W, n, n);
7
8  cublasSetVector(n, sizeof(*nvec), nvec, 1, c_nvec, 1);
9
10 // Formula for maximizing the normal distribution
11 // using the tangent function
12 // m - number of samples
13 // G1, G2 -- values of tangent functions
14 // c_X - input values
15 // c_hypTan - matrix with results from kernel function
16 // nvec - vector with results from kernel funtion
17
18 tanhGPU_wrapper(c_hypTan, a1, n, m, c_X, c_W, c_G1, c_nvec);
19 cublasGetVector(nn, sizeof(*c_G1), c_G1, 1, G1, 1);
20 cublasGetVector(n, sizeof(*c_nvec), c_nvec, 1, nvec,1);
21
22 #pragma omp parallel // efficiency: 95\%,
23                      // gain: 1,90x according to Intel Advisor
24 {
25 #pragma omp for
26   for(int i = 0; i < n; i++)
```

```
27      G2[i*n:n] = W[i*n:n]*nvec[i];
28  } //parallel
29
30  // Modification of weights
31  W[0:nn] = (G1[0:nn] - G2[0:nn])/m*a1;
```

3. Results

All tests performed on these architectures are described in Table 1.

Table 1
Selected architectures

Infrastructure	Cluster name	CPU	Number of cores	GPU
UMCS	Solaris	2x Intel Xeon E5-2670 v3 @ 2.30GHz	24	NVIDIA Tesla V100s
PLGrid	Zeus	2x Intel Xeon X5650 @ 2.67GHz	12	NVIDIA Tesla M2090
PLGrid	Prometheus	2x Intel Xeon E5-2680 @ 2.5GHz	24	NVIDIA Tesla K40d

The study compared the speed of the fastICA implementation for the data sets of 256×1000 (1 second of recording), 256×10000 (10 seconds of recording), 256×100000 (100 seconds of recording) and 256×1000000 (1000 seconds of recording) data sets. For each architecture, the time efficiency was compared depending on the number of threads the program was launched with. The initial selection of weights was the same each time, and the resulting time was averaged over the number of needed iterations (for the purposes of this study, it was established that approximately 100 iterations were needed for the selected EEG data).

3.1. Tests of implementation

The previous research proved that the use of virtual cores does not speed up the performance of the algorithm as is the case with physical cores, so the tests were performed without the use of hyper-threading. [6, 7, 9, 10].

3.1.1. Solaris

Table 2 shows the program execution time in seconds for all data. Figure 2 presents the acceleration obtained by the CPU+GPU version compared to the CPU version with a given number of threads. As follows from the results the use of GPU capabilities speeds up the performance of the algorithm. The difference is not as significant as the number of threads increases, but there is still a slight time gain from using the extra CPU cores.

Table 2

Solaris: 2x Intel Xeon CPU E5-2670 v3 @ 2.30GHz – 24 cores, NVIDIA Tesla V100s

	256x1000		256x10000		256x100000		256x1000000	
Number of threads	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**
1	3.075	2.781	4.118	11.577	5.100	100.253	17.346	989.861
2	3.197	2.582	4.150	8.154	5.253	64.152	15.541	648.398
3	3.346	2.456	4.479	7.572	4.860	57.090	14.205	571.254
4	2.862	2.087	3.835	6.379	4.337	48.028	13.069	461.153
5	2.827	2.338	3.942	6.511	5.107	49.054	13.467	459.481
6	3.026	1.992	3.813	5.738	4.571	41.999	13.278	409.896
7	2.934	1.958	3.944	5.919	4.358	45.687	12.802	404.973
8	2.721	1.918	3.716	5.291	4.530	39.361	12.306	382.040
9	2.982	2.039	4.237	5.643	4.486	42.711	12.386	390.353
10	2.828	1.977	4.213	5.440	4.294	38.645	12.465	364.328
11	2.879	2.003	3.946	5.603	4.666	40.752	12.361	375.422
12	2.880	1.925	3.783	5.448	4.429	40.369	12.562	354.141
13	2.799	1.976	3.887	5.379	4.352	42.271	12.426	368.260
14	2.861	1.981	4.246	5.256	4.468	38.177	12.124	346.272
15	2.886	1.925	3.804	5.474	4.429	40.127	12.181	350.959
16	2.705	1.821	4.008	5.215	4.099	36.894	12.385	341.640
17	2.948	1.997	3.886	5.658	4.415	38.647	12.524	348.708
18	2.890	2.013	3.938	5.347	4.392	37.018	12.099	340.452
19	2.903	2.029	3.946	5.496	4.423	40.451	12.392	341.226
20	3.067	2.066	3.802	5.360	4.443	37.483	12.184	351.473
21	2.900	1.949	3.958	5.435	4.403	37.311	12.310	339.592
22	2.837	2.076	4.060	5.471	4.357	38.132	12.012	342.490
23	2.998	2.124	3.904	5.475	4.422	39.912	12.412	341.577
24	3.085	2.106	4.052	5.439	4.450	37.579	12.189	346.825

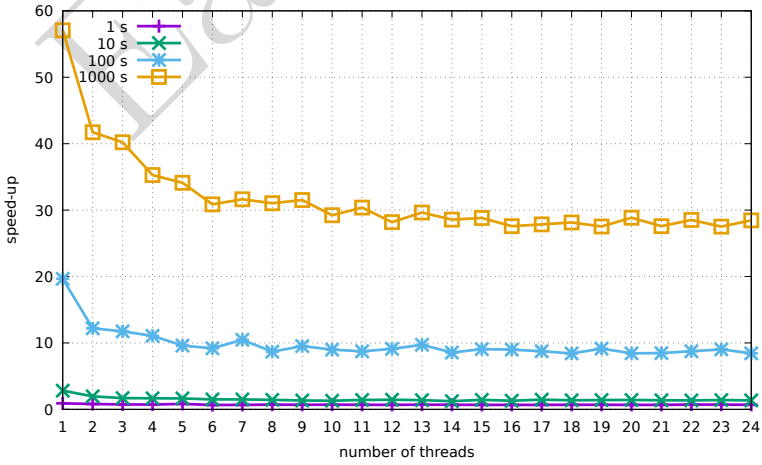
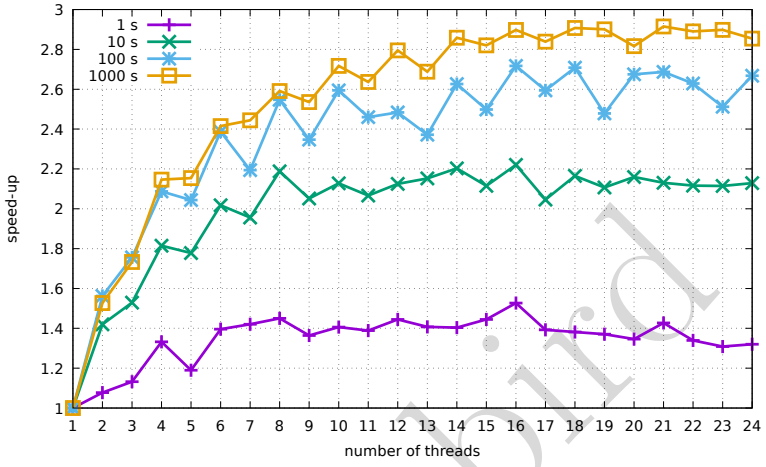
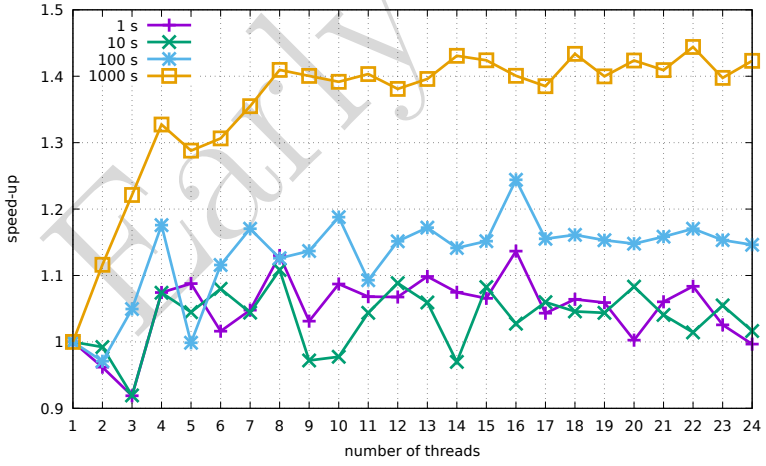


Figure 2. Solaris: 2x Xeon CPU E5-2670 v3 @ 2.30GHz – 24 cores, NVIDIA Tesla V100s – comparing the acceleration of the CPU+GPU version to the CPU

Figure 3 shows the acceleration of the CPU version (a) and CPU+GPU (b) depending on the number of threads. The speedup is calculated in relation to a single-threaded program. They show more clearly that adding computational cores still generates a profit, although in the case of the CPU version the graphs are flatter.



(a)



(b)

Figure 3. Solaris: 2x Intel Xeon CPU E5-2670 v3 @ 2.30GHz – 24 cores, NVIDIA Tesla V100s – the acceleration compared to the single-threaded version: a) CPU-based implementation; b) CPU+GPU-based implementation

Notably, the machine with the Tesla V100S card takes 17 seconds for the largest data size using a single thread. However, we are to do with very powerful equipment of

the latest generation. It is worth noting, however, that the use of CUDA capabilities without introducing parallel calculations on the CPU reduces the calculation time significantly. It is important in the context of building computing units in the research centers for the processing of EEG data. Providing such a unit with a graphics card with adequate power can be cheaper than building a multi-core computing cluster.

3.1.2. Zeus and Prometheus

Tables 3 and 4 show the program execution time in seconds for all data. As in the Solaris cluster, more cores reduce the computation time, the implementation is similarly scalable, and the increase in data size generates better time gains.

Table 3
Zeus: 2x Intel Xeon X5650 @ 2.67GHz – 12 cores, Tesla M2090

	256x1000		**256x10000**		**256x100000**		**256x1000000**	
Number of threads	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**
1	8.596	8.824	6.498	21.854	13.592	224.053	109.585	2242.188
2	6.994	6.436	5.408	13.451	12.222	130.692	101.576	1298.195
3	6.134	5.070	4.937	10.122	11.638	98.258	98.350	970.889
4	5.476	3.963	4.632	7.903	10.985	77.353	96.650	765.494
5	5.623	3.923	4.674	7.578	10.957	71.673	96.109	704.106
6	5.513	3.748	4.585	6.679	10.880	63.229	95.326	626.615
7	5.522	3.704	4.502	6.583	10.791	61.962	95.154	610.624
8	5.190	3.304	4.454	6.055	10.723	56.966	94.769	555.167
9	5.436	3.603	4.459	6.303	10.664	57.433	94.745	555.466
10	5.411	3.564	4.419	6.299	10.710	57.180	94.519	543.231
11	5.366	3.716	4.455	6.135	10.802	56.176	94.573	545.689
12	5.573	3.506	4.551	6.102	10.571	57.342	94.395	552.920

Table 4
Prometheus: 2x Intel Xeon E5-2680 @ 2.5GHz – 24 cores, Tesla K40d

	256x1000		**256x10000**		**256x100000**		**256x1000000**	
Number of threads	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**
1	2.494	2.575	1.470	6.272	4.328	63.333	43.934	633.295
2	2.144	2.340	1.448	4.237	4.184	40.630	42.545	399.542
3	2.057	2.223	1.459	3.724	4.154	33.801	42.229	353.096
4	1.968	2.056	1.405	3.242	4.059	29.936	41.688	300.085
5	2.089	2.150	1.421	3.299	4.099	28.971	41.608	281.807
6	1.951	2.083	1.366	3.091	4.029	26.651	41.509	259.350
7	1.911	2.042	1.352	3.073	4.064	28.096	41.478	271.448
8	1.733	1.937	1.305	2.798	4.026	24.679	41.283	237.445
9	1.934	1.956	1.343	2.985	3.967	25.829	41.204	246.441

Table 4 cont.

	256x1000		256x10000		256x100000		256x1000000	
Number of threads	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**	**CPU+ GPU**	**CPU**
11	1.866	1.899	1.338	2.819	3.970	24.460	41.245	232.758
12	1.869	1.917	1.409	2.737	3.965	24.172	41.082	216.436
13	1.885	1.941	1.285	2.836	3.935	23.769	41.180	234.071
14	1.879	1.864	1.319	2.783	4.006	23.355	41.048	223.344
15	1.907	2.054	1.387	2.857	3.968	24.689	41.112	215.137
16	1.912	1.861	1.318	2.803	4.002	22.574	41.096	207.854
17	1.871	1.874	1.371	2.937	4.019	24.664	40.990	241.892
18	1.892	1.993	1.377	2.854	3.991	22.576	41.045	219.282
19	1.940	2.132	1.330	2.920	3.957	25.215	41.052	227.162
20	1.975	2.142	1.449	2.828	4.159	23.150	40.995	205.798
21	2.103	2.042	1.335	2.992	3.936	23.164	41.055	216.180
22	1.959	1.954	1.316	2.924	3.921	22.844	40.982	204.775
23	1.975	2.167	1.385	3.063	3.982	24.487	41.162	226.745
24	2.243	2.080	1.483	2.893	3.992	22.750	41.086	213.694

Figures 5 and 4 present the acceleration of the CPU+GPU version compared to that of CPU with a given number of threads. Figures 6 and 7 show the obtained acceleration of the version of the implementation using CPU (a) and CPU+GPU (b) depending on the number of threads. The speedup is calculated in relation to a single-threaded program. One can see a similar relationship, that is, using more threads, you can still expect a gain in time.

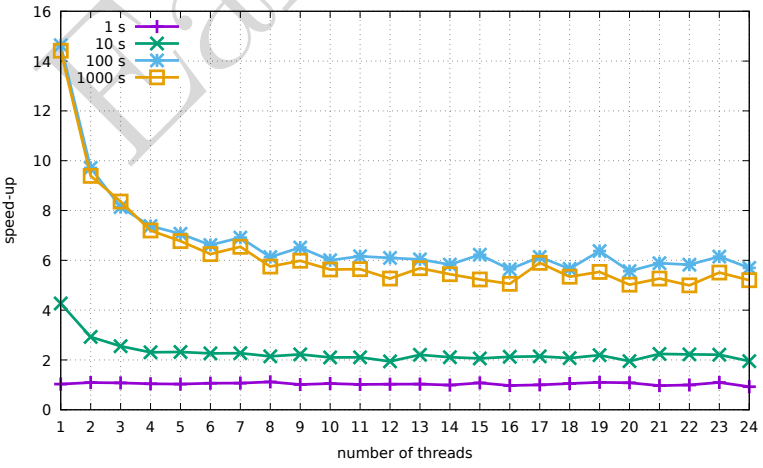


Figure 4. Prometheus: 2x Intel Xeon E5-2680 @ 2.5GHz – 24 cores, Tesla K40d – comparing the acceleration of the CPU+GPU version to the CPU

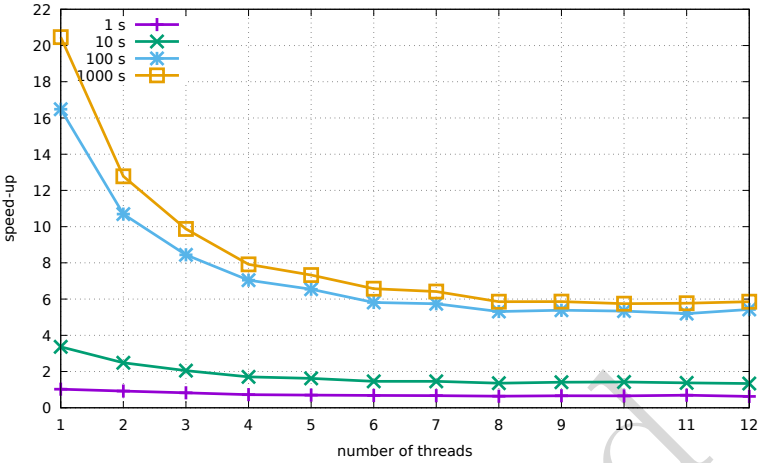
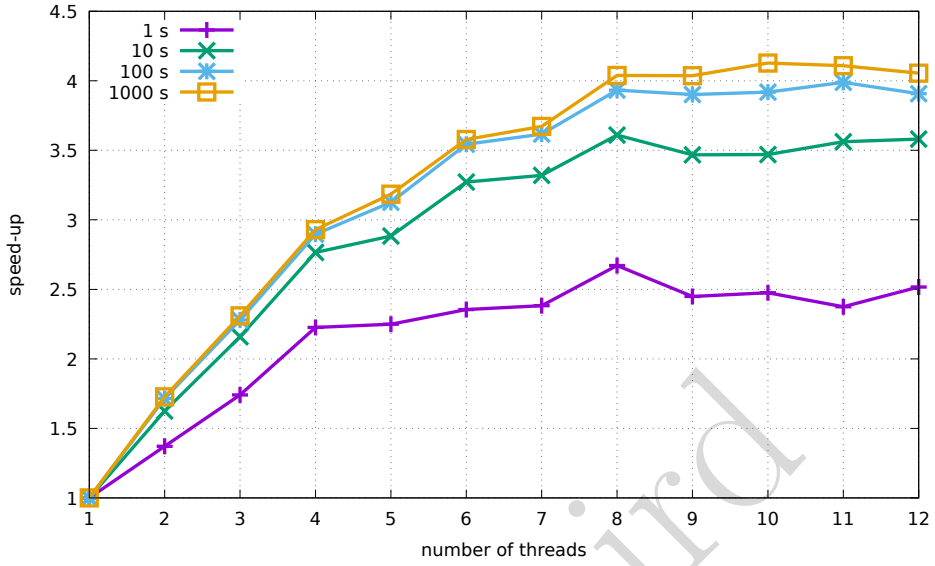
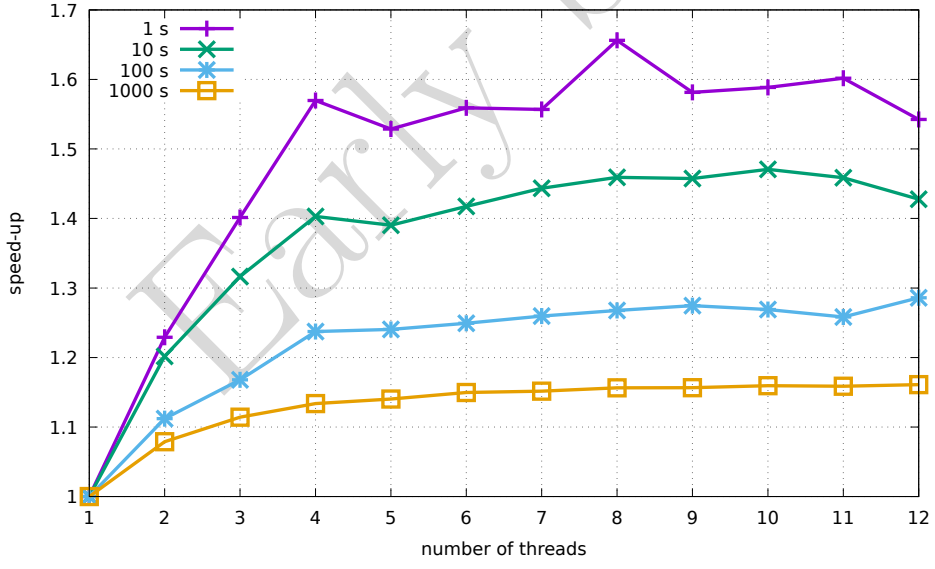


Figure 5. Zeus: 2x Intel Xeon X5650 @ 2.67GHz – 12 cores, Tesla M2090 – comparing the acceleration of the CPU+GPU version to the CPU

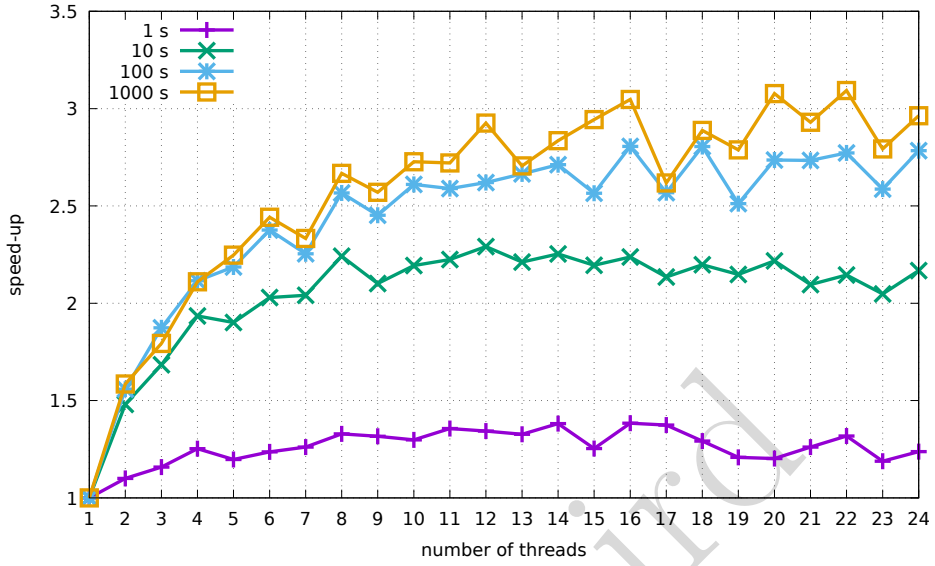


(a))CPU-based implementation

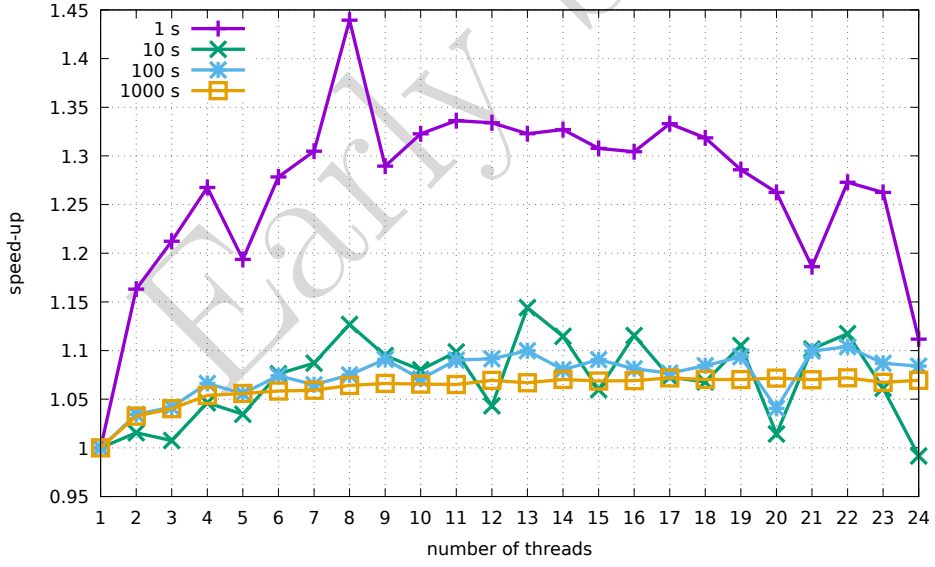


(b))CPU+GPU-based implementation

Figure 6. Zeus: 2x Intel Xeon X5650 @ 2.67GHz – 12 cores, Tesla M2090 – the acceleration compared to the single-threaded version



(a))CPU-based implementation



(b))CPU+GPU-based implementation

Figure 7. Prometheus: 2x Intel Xeon E5-2680 @ 2.5GHz – 24 cores, Tesla K40d – the acceleration compared to the single-threaded version

4. Discussion

The results proved that the support from the graphics card and CUDA technology accelerates the execution time of the algorithm significantly. The increase in the number of CPU cores still resulted in acceleration, although the benefits were not so visible.

Owing to the variety of architectures, it is possible to indicate which elements of the implementation are crucial for time optimization (in terms of reducing the running time) of the algorithm. At the same time, it is worth mentioning that due to the complexity of the structure of computing clusters, creating efficient programs on them is a demanding task [14,20,21,28]. The fact that fastICA is an iterative method requires thread synchronization after each iteration, which affects the execution time significantly and makes it difficult to scale. It can be stated that this is the main "bottleneck" of the algorithm with no prospects for improving the results at this point.

As a matter of fact, the machine with the NVIDIA Tesla V100s graphics card gave the most satisfactory results, although cards of this type are not the cheapest product. However, the latest generation Intel Xeon processors are also quite expensive. These factors should also be considered when building a computational machine for EEG data analysis. Providing such unit with a graphics card with the appropriate power could be less expensive than constructing a multi-core computing cluster.

5. Conclusions and future works

This paper presents the time execution results for a parallel version of the fastICA algorithm adapted to EEG signals and multi-core architectures (capacity of Intel architectures and available libraries as well as extensions together with CUDA libraries). Tests were carried out on several computational clusters using the EEG data of various sizes. The paper indicates the possibility of improving the parallelized version of the algorithm by transferring more calculations to the graphics card.

The future plan is also to integrate existing solutions with EGI System NetStation [6–10] and after separation of sources, to make an attempt to reject artifacts found in this way automatically. A possible method of doing this is to use, among others, convolutional neural networks.

Neuroimaging techniques, including EEG, provide excellent opportunities to get to know how a person functions, and thus, based on the research and diagnostics, to improve living conditions. Shortening the time of the electroencephalographic signal analysis tools, it is possible to help more patients, accelerate research, and improve BCI solutions [5,15,16,19,25,27].

Acknowledgements

This research was partially supported by PLGrid Infrastructure.

References

[1] Albajes-Eizagirre A., Dubreuil Vall L., David I.S., Riera A., Soria-Frisch A., Dunne S., Ruffini G.: *EEG/ERP analysis: methods and applications*, CRC Press, 2014.

[2] Brown G.D., Yamada S., Sejnowski T.J.: Independent component analysis at the neural cocktail party, *Trends in neurosciences*, vol. 24(1), pp. 54–63, 2001.

[3] Delorme A., Sejnowski T., Makeig S.: Enhanced detection of artifacts in EEG data using higher-order statistics and independent component analysis, *Neuroimage*, vol. 34(4), pp. 1443–1449, 2007.

[4] Dickter C.L., Kieffaber P.D.: *EEG methods for the psychological sciences*, SAGE Knowledge, Los Angeles, 2014.

[5] Duch W., Nowak W., Meller J., Osiński G., Dobosz K., Mikołajewski D., Wójcik G.M.: Computional approach to understanding autism spectrum disorders, *Computer Science*, vol. 13(2), p. 47, 2012. doi: 10.7494/csci.2012.13.2.47.

[6] Gajos A., Wójcik G.M.: Independent component analysis of EEG data for EGI system, *Bio-Algorithms and Med-Systems*, vol. 12(2), pp. 67–72, 2016.

[7] Gajos-Balińska A., Wójcik G.M., Stpiczyński P.: Concept of independent component analysis algorithm parallelisation. In: *Proceedings of Cracow Grid Workshop*, pp. 55–56, CGW'15, 2015.

[8] Gajos-Balińska A., Wójcik G.M., Stpiczyński P.: Parallel independent component analysis algorithm - performance comparison for EEG signal. In: *Proceedings of Cracow Grid Workshop*, CGW'17, 2017.

[9] Gajos-Balińska A., Wójcik G.M., Stpiczyński P.: High performance optimization of independent component analysis algorithm for EEG data, *Lecture Notes in Computer Science*, vol. 10777, pp. 495–504, 2018.

[10] Gajos-Balińska A., Wójcik G.M., Stpiczyński P.: Cooperation of CUDA and Intel multi-core architecture in the independent component analysis algorithm for EEG data, *Bio-Algorithms and Med-Systems*, vol. 16(3), 2020.

[11] Hyvarinen A.: Fast and robust fixed-point algorithms for independent component analysis, *IEEE Transactions on Neural Networks*, vol. 10(3), pp. 626–634, 1999.

[12] Hyvärinen A., Oja E.: Independent component analysis: algorithms and applications, *Neural networks*, vol. 13(4), pp. 411–430, 2000.

[13] Kawala-Janik A., Bauer W., Al-Bakri A., Haddix C., Yuvaraj R., Cichon K., Podraza W.: Implementation of low-pass fractional filtering for the purpose of analysis of electroencephalographic signals. In: *Conference on Non-integer Order Calculus and Its Applications*, pp. 63–73, Springer, 2017.

[14] Lastovetsky A., Szustak L., Wyrzykowski R.: Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28(3), pp. 787–797, 2016.

[15] Mikołajewska E., Mikołajewski D.: Integrated IT environment for people with disabilities: a new concept, *Open Medicine*, vol. 9(1), pp. 177–182, 2014.

[16] Mikołajewska E., Mikołajewski D.: The prospects of brain–computer interface applications in children, *Open Medicine*, vol. 9(1), pp. 74–79, 2014.

[17] NetStation acquisition technical manual. Documentation, EGI, 2011.

[18] Rahman R.: *Intel Xeon Phi coprocessor architecture and tools: the guide for application developers*, Apress, Berkely, CA, USA, 2013.

[19] Rojek I., Macko M., Mikolajewski D., Saga M., Burczynski T.: Modern methods in the field of machine modelling and simulation as a research and practical issue related to Industry 4.0, *The Bulletin of the Polish Academy of Sciences Technical Sciences*, vol. 69(2), 2021. doi: 10.24425/bpasts.2021.136717.

[20] Szustak L.: Strategy for data-flow synchronizations in stencil parallel computations on multi-/manycore systems, *The Journal of Supercomputing*, vol. 74(4), pp. 1534–1546, 2018.

[21] Szustak L., Bratek P.: Performance portable parallel programming of heterogeneous stencils across shared-memory platforms with modern Intel processors, *The International Journal of High Performance Computing Applications*, vol. 33(3), pp. 534–553, 2019.

[22] Tadeusiewicz R., Śmiałowska M., Hess G., Błaszczyk J., Kamiński W.A., Lazarewicz M.T., Strzelecki M., Wójcik G.M.: *Neurocybernetyka teoretyczna*, Wydawnictwa Uniwersytetu Warszawskiego, 2009.

[23] Ungureanu M., Bigan C., Strungaru R., Lazarescu V.: Independent component analysis applied in biomedical signal processing, *Measurement Science Review*, vol. 4(2), p. 18, 2004.

[24] Wojcik G.: *Selected methods of quantitative analysis in electroencephalography*, pp. 35–54, De Gruyter, 2020. doi: 10.1515/9783110667219-003.

[25] Wojcik G.M., Masiak J., Kawiak A., Kwasniewicz L., Schneider P., Polak N., Gajos-Balinska A.: Mapping the Human Brain in Frequency Band Analysis of Brain Cortex Electroencephalographic Activity for Selected Psychiatric Disorders, *Frontiers in Neuroinformatics*, vol. 12, 2018. doi: 10.3389/fninf.2018.00073.

[26] Wojcik G.M., Masiak J., Kawiak A., Kwasniewicz L., Schneider P., Postepski F., Gajos-Balinska A.: Analysis of Decision-Making Process Using Methods of Quantitative Electroencephalography and Machine Learning Tools, *Frontiers in Neuroinformatics*, vol. 13, 2019. doi: 10.3389/fninf.2019.00073.

[27] Wojcik G.M., Masiak J., Kawiak A., Schneider P., Kwasniewicz L., Polak N., Gajos-Balinska A.: New Protocol for Quantitative Analysis of Brain Cortex Electroencephalographic Activity in Patients With Psychiatric Disorders, *Frontiers in Neuroinformatics*, vol. 12, 2018. doi: 10.3389/fninf.2018.00027.

[28] Wyrzykowski R., Szustak L., Rojek K.: Parallelization of 2D MPDATA EULAG algorithm on hybrid architectures with GPU accelerators, *Parallel Computing*, vol. 40(8), pp. 425–447, 2014.

Affiliations

Anna Gajos-Balińska
Department of Neuroinformatics and Biomedical Engineering, Institute of Computer Science, Maria Curie-Sklodowska University, ul. Akademicka 9, 20-033 Lublin, Poland, anna.gajos-balinska@umcs.pl

Grzegorz M. Wójcik
Department of Neuroinformatics and Biomedical Engineering, Institute of Computer Science, Maria Curie-Sklodowska University, ul. Akademicka 9, 20-033 Lublin, Poland

Przemysław Stpiczyński
Department of Information Systems Software, Institute of Computer Science, Maria Curie-Sklodowska University, ul. Akademicka 9, 20-033 Lublin, Poland

Received: 05.06.2023
Revised: 29.11.2023
Accepted: 29.11.2023